

Design of Digital Circuits

Lecture 7.1: Sequential Logic Design II

Prof. Onur Mutlu

ETH Zurich

Spring 2019

14 March 2019

Agenda for This Week

■ Today

- Wrap up Sequential Logic
- Hardware Description Languages and Verilog
 - Combinational Logic
 - Sequential Logic

■ Tomorrow

- Timing and Verification

Agenda for Next Week

■ Thursday

- Von Neumann Model of Execution
- Instruction Set Architecture
 - LC-3 and MIPS

■ Friday

- ISA and Assembly Programming

Extra Assignment 1: Lecture Video

- Why study computer architecture?
- Why is it important?
- **Future Computing Architectures**
- **Required Assignment**
 - **Watch** my inaugural lecture at ETH and understand it
 - <https://www.youtube.com/watch?v=kgiZISOcGFM>
- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page summary** of the lecture
 - What are your key takeaways?
 - What did you learn?
 - What did you like or dislike?
 - Upload PDF file to Moodle – Deadline: Friday, March 15.

Extra Assignment 2: Moore's Law (I)

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965
- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page review**
 - Upload PDF file to Moodle – Deadline: Friday, March 22
- I strongly recommend that you **follow my guidelines for (paper) review** (see next slide)

Extra Assignment 2: Moore's Law (II)

■ Guidelines on how to review papers critically

- ❑ **Guideline slides:** [pdf](#) [ppt](#)
- ❑ **Video:** <https://www.youtube.com/watch?v=tOL6FANAj8c>
- ❑ Example reviews on “Main Memory Scaling: Challenges and Solution Directions” ([link to the paper](#))
 - [Review 1](#)
 - [Review 2](#)
- ❑ Example review on “Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems” ([link to the paper](#))
 - [Review 1](#)

Required Readings (This Week)

- Hardware Description Languages and Verilog
 - H&H Chapter 4 in full
- Timing and Verification
 - H&H Chapters 2.9 and 3.5 + (start Chapter 5)
- By tomorrow, make sure you are done with
 - **P&P Chapters 1-3 + H&H Chapters 1-4**

Required Readings (Next Week)

- Von Neumann Model, LC-3, and MIPS
 - P&P, Chapter 4, 5
 - H&H, Chapter 6
 - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
 - H&H, Appendix B (MIPS instructions)

- Programming
 - P&P, Chapter 6

- **Recommended:** Digital Building Blocks
 - H&H, Chapter 5

Wrap-Up Sequential Logic Circuits and Design

Circuits that Can Store Information

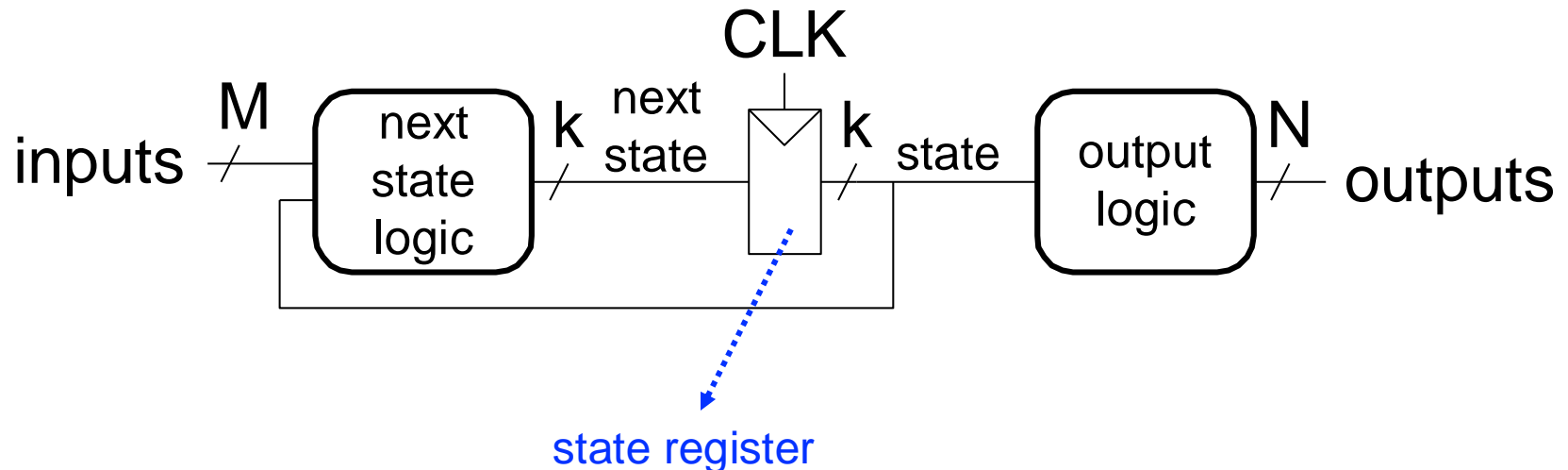
The Gated D Latch

Sequential Logic Circuits

Review: Finite State Machines

Recall: Finite State Machines (FSMs)

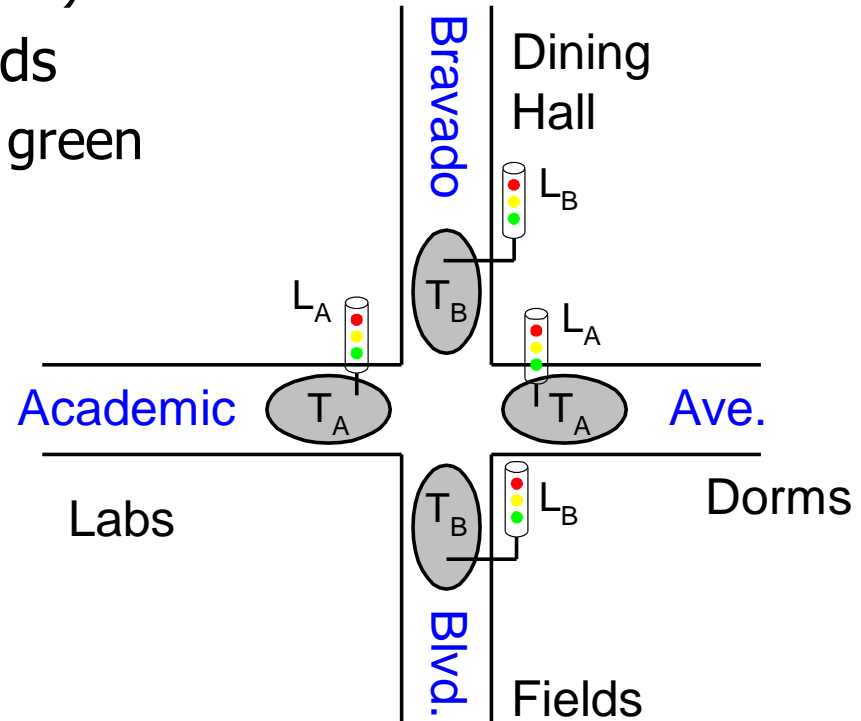
- Each FSM consists of three separate parts:
 - next state logic
 - state register
 - output logic



At the beginning of the clock cycle, next state is latched into the state register

Recall: Finite State Machine Example

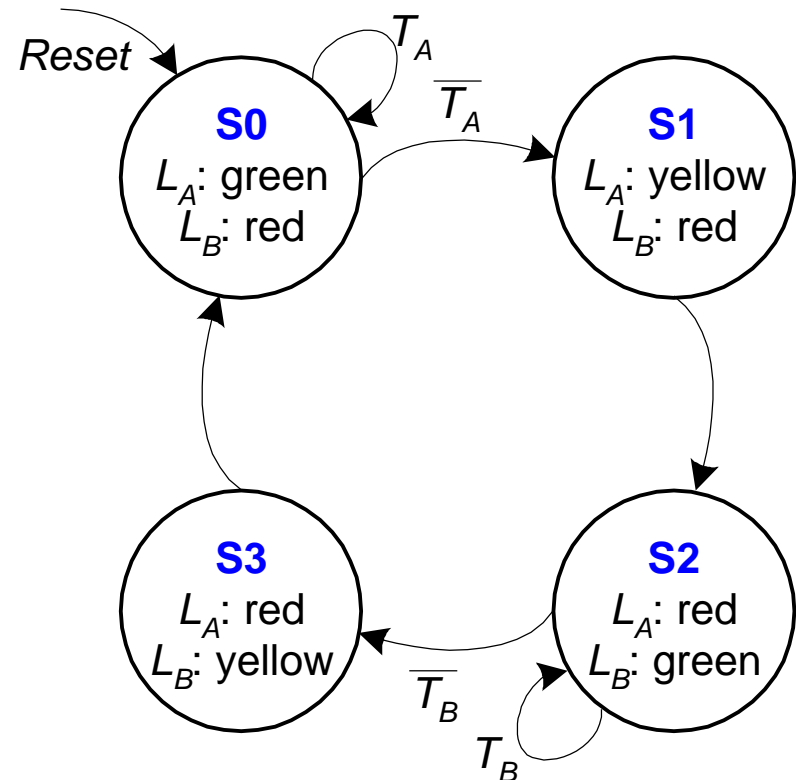
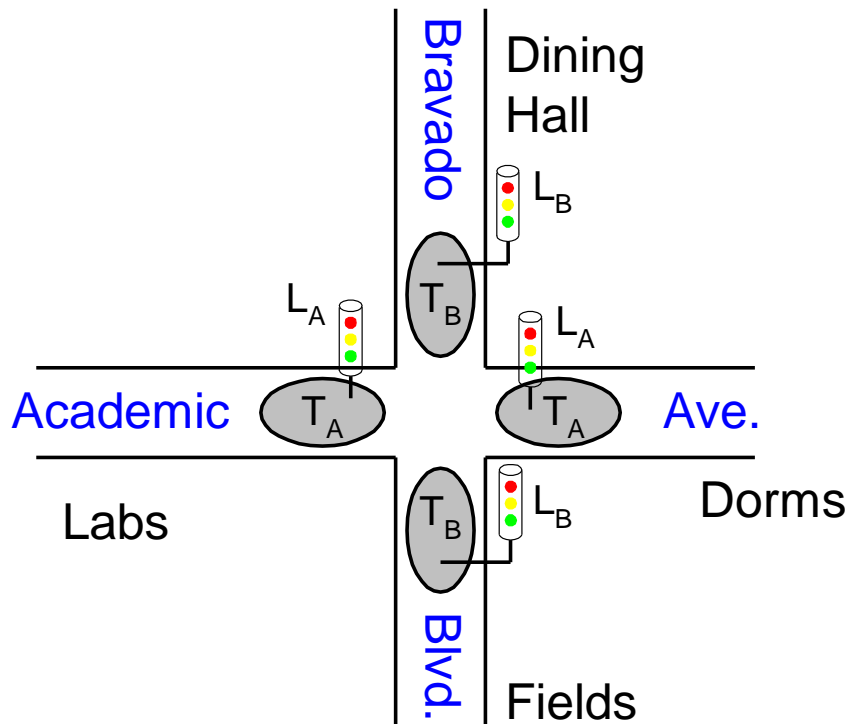
- “Smart” traffic light controller
 - **2 inputs:**
 - Traffic sensors: T_A , T_B (TRUE when there's traffic)
 - **2 outputs:**
 - Lights: L_A , L_B (Red, Yellow, Green)
 - State can change every 5 seconds
 - Except if green and traffic, stay green



From H&H Section 3.4.1

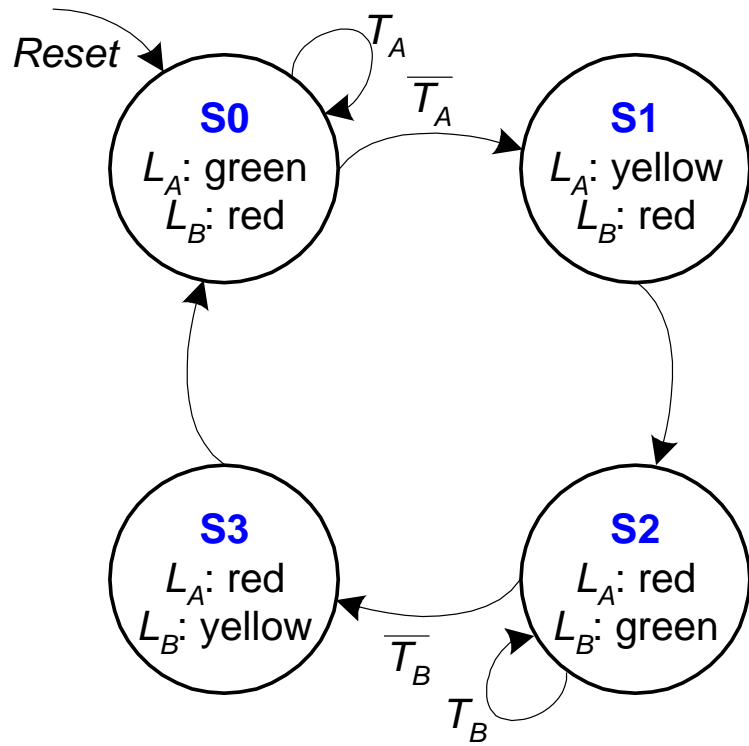
Recall: FSM Transition Diagram

- **Moore FSM:** outputs labeled in each state
 - **States:** Circles
 - **Transitions:** Arcs



Recall: Finite State Machine: State Transition Table

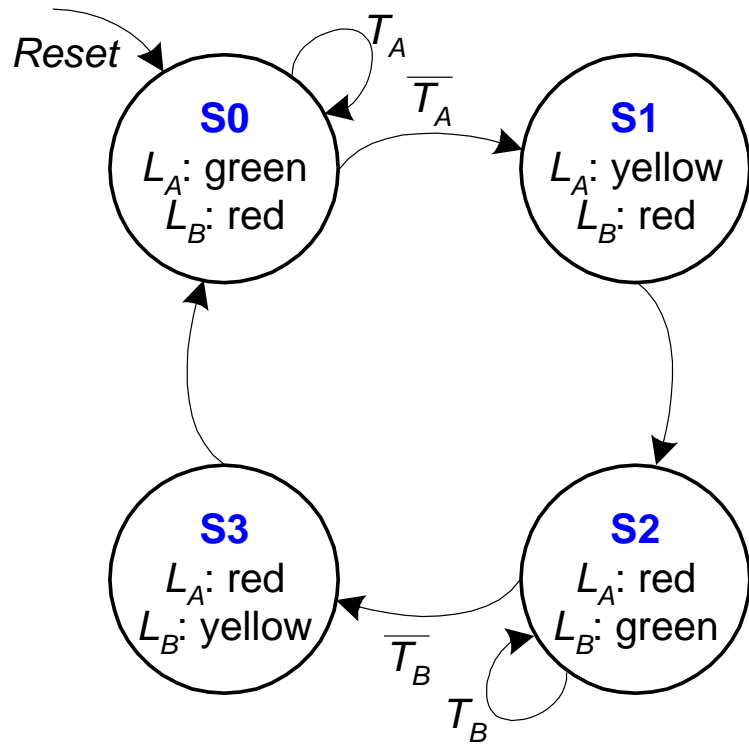
Recall: FSM State Transition Table



Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

Recall: FSM State Transition Table



Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

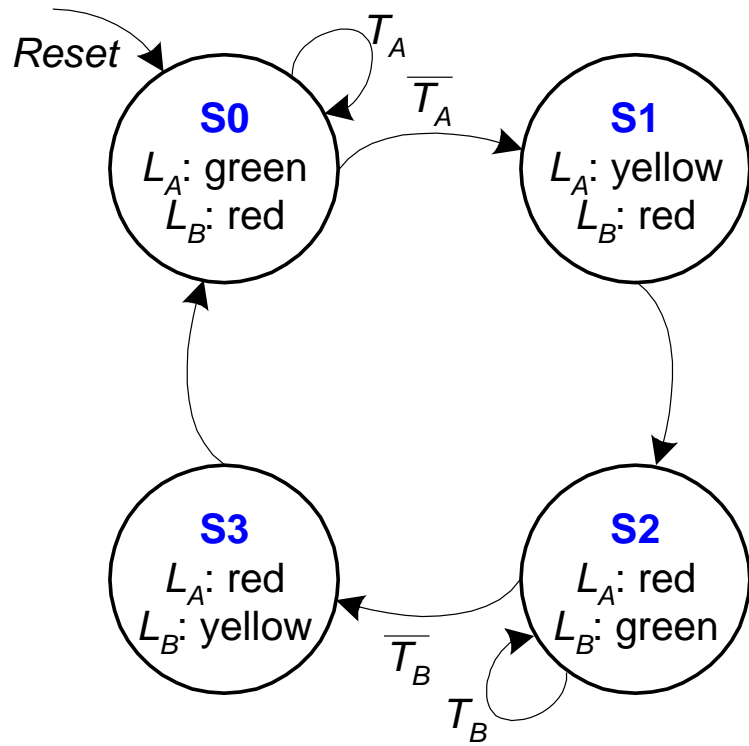
State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = S_1 \text{ xor } S_0 \quad \textbf{(Simplified)}$$

$$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

Recall: Finite State Machine: Output Table

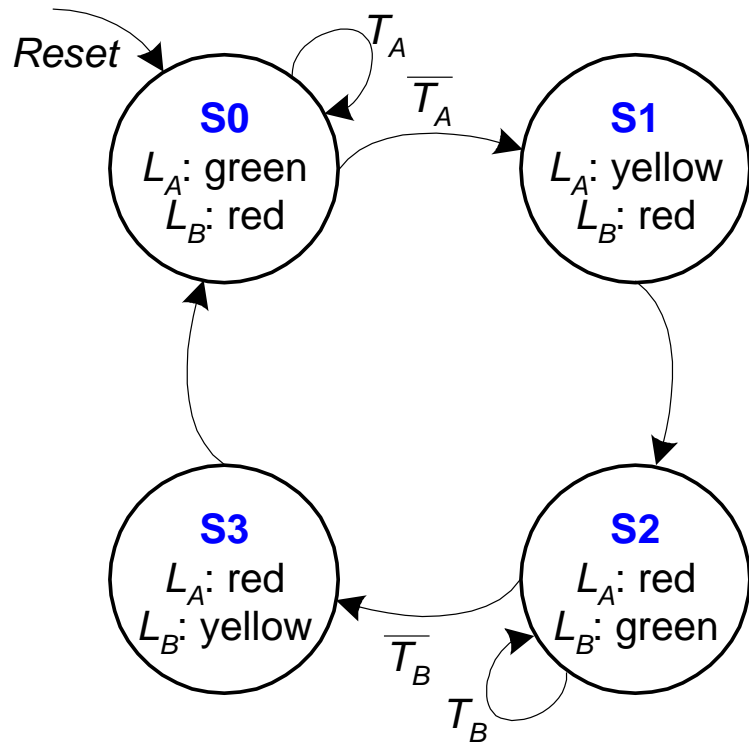
Recall: FSM Output Table



Current State		Outputs	
S_1	S_0	L_A	L_B
0	0	green	red
0	1	yellow	red
1	0	red	green
1	1	red	yellow

Output	Encoding
green	00
yellow	01
red	10

Recall: FSM Output Table



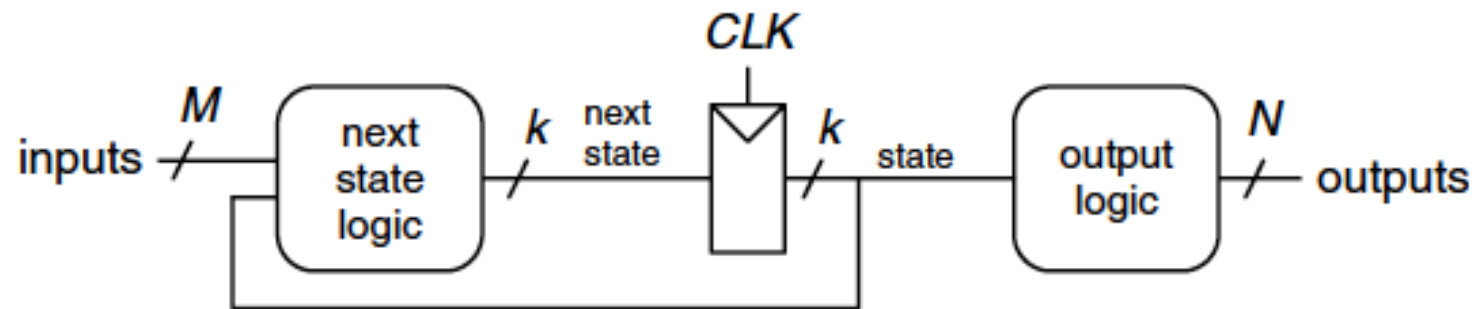
Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned}
 L_{A1} &= S_1 \\
 L_{A0} &= \overline{S_1} \cdot S_0 \\
 L_{B1} &= \overline{S_1} \\
 L_{B0} &= S_1 \cdot S_0
 \end{aligned}$$

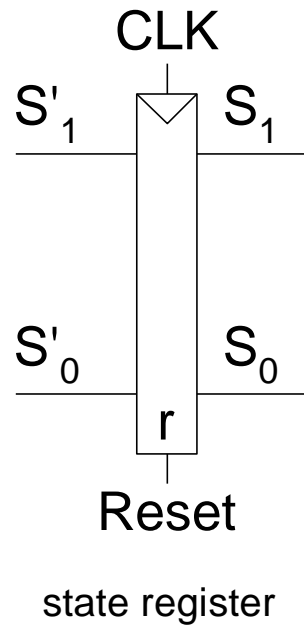
Output	Encoding
green	00
yellow	01
red	10

Recall: Finite State Machine: Schematic

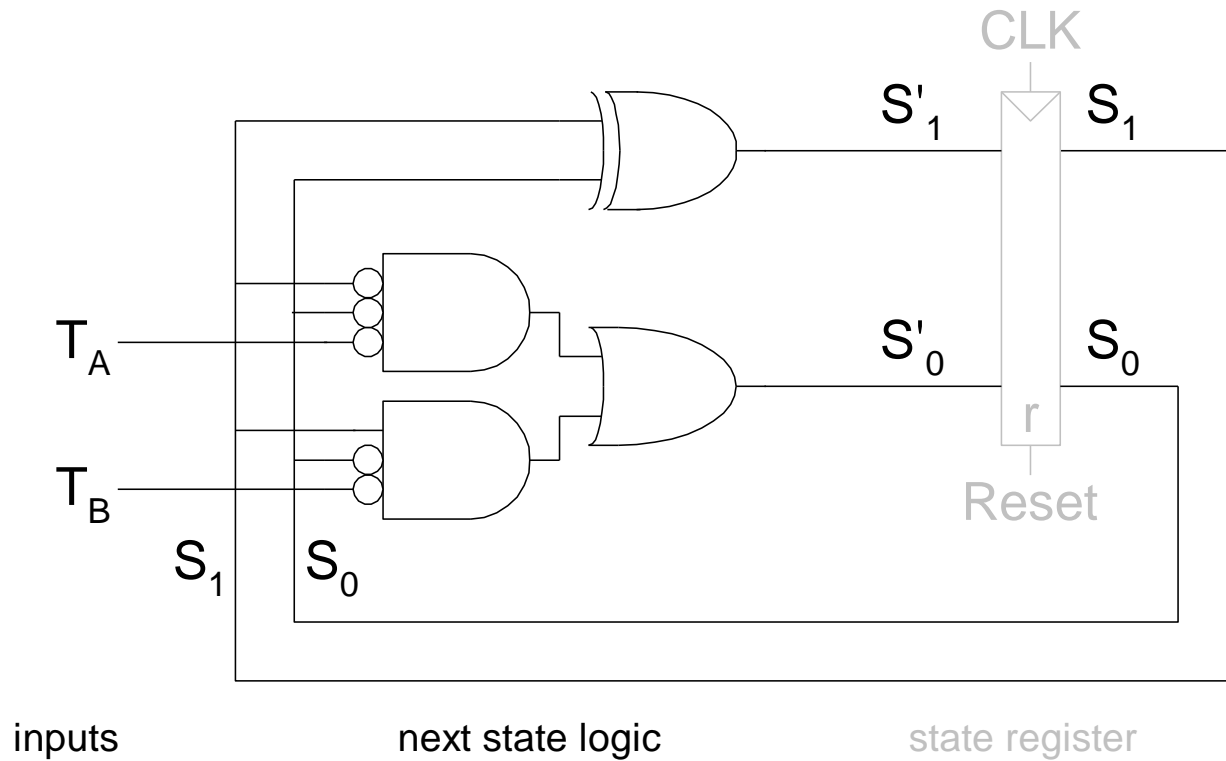
Recall: FSM Schematic: State Register



Recall: FSM Schematic: State Register



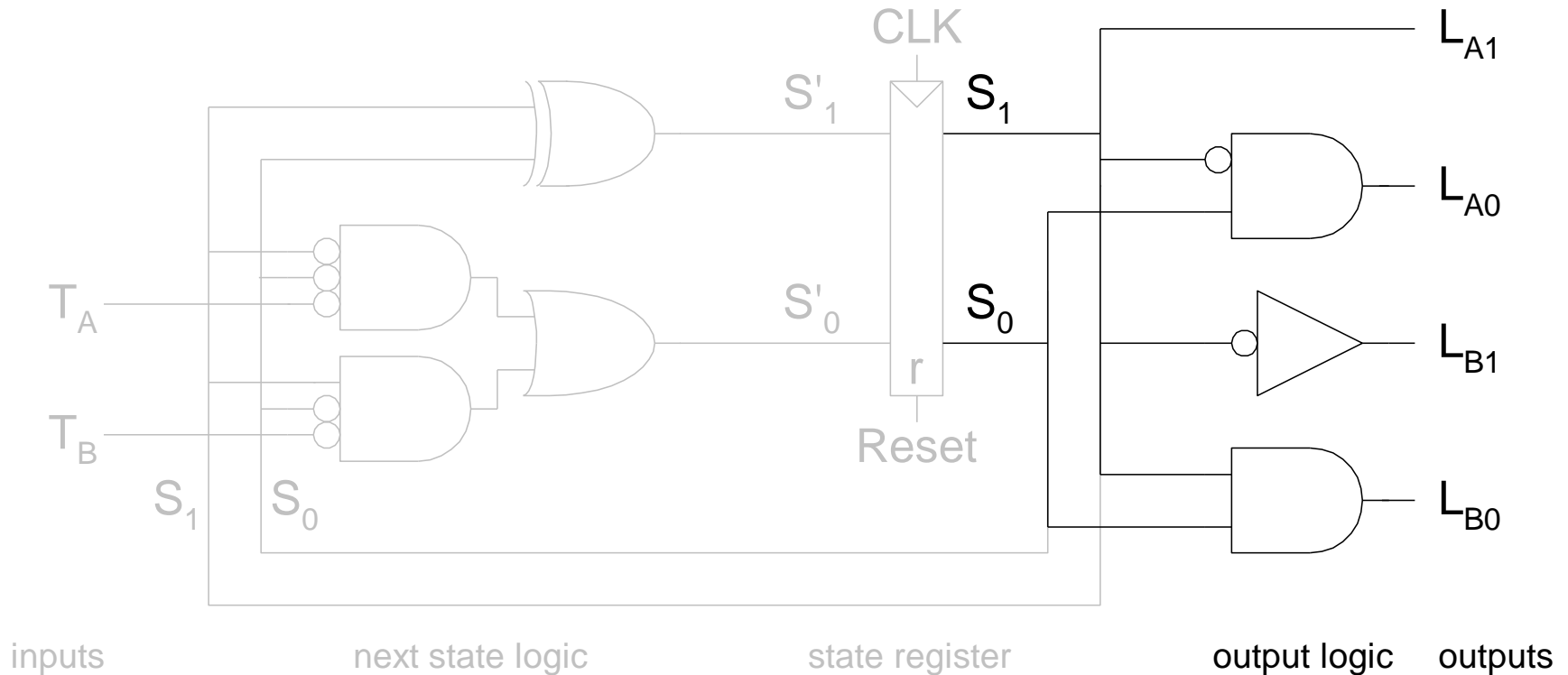
Recall: FSM Schematic: Next State Logic



$$S'_1 = S_1 \text{ xor } S_0$$

$$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

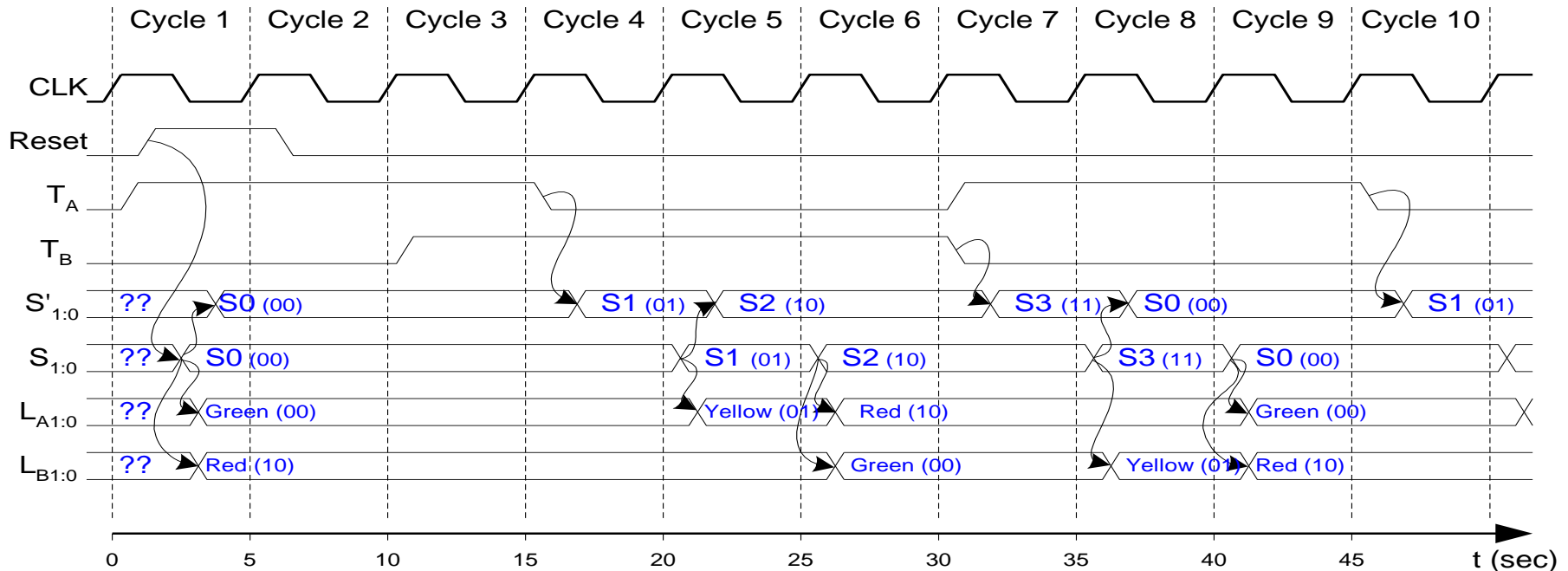
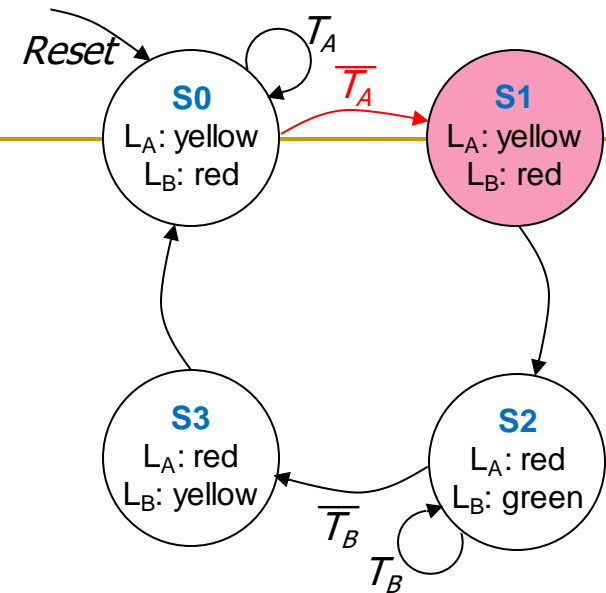
Recall: FSM Schematic: Output Logic



$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S_1} \cdot S_0 \\ L_{B1} &= \overline{S_1} \\ L_{B0} &= S_1 \cdot S_0 \end{aligned}$$

Recall: FSM Timing

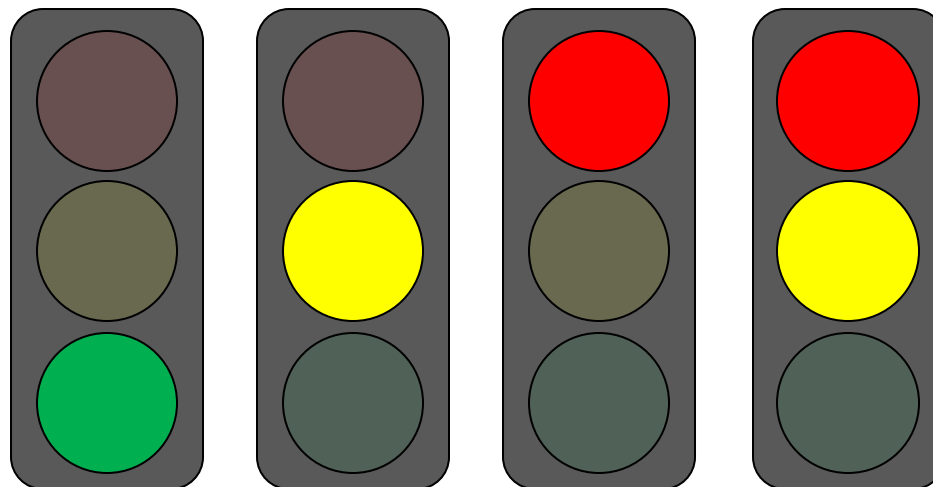
See H&H Chapter 3.4



Finite State Machine: State Encoding

FSM State Encoding

- How do we encode the state bits?
 - Three common state binary encodings with different tradeoffs
 1. **Fully Encoded**
 2. **1-Hot Encoded**
 3. **Output Encoded**
- Let's see an example **Swiss** traffic light with 4 states
 - Green, Yellow, Red, Yellow+Red



FSM State Encoding (II)

1. Binary Encoding (Full Encoding):

- ❑ Use the minimum number of bits used to encode all states
 - Use $\log_2(num_states)$ bits to represent the states
- ❑ *Example states:* 00, 01, 10, 11
- ❑ **Minimizes** # flip-flops, but not necessarily output logic or next state logic

2. One-Hot Encoding:

- ❑ Each bit encodes a different state
 - Uses num_states bits to represent the states
 - Exactly 1 bit is "hot" for a given state
- ❑ *Example states:* 0001, 0010, 0100, 1000
- ❑ **Simplest design process** – very automatable
- ❑ **Maximizes** # flip-flops, **minimizes** next state logic

FSM State Encoding (III)

3. Output Encoding:

- ❑ Outputs are **directly accessible** in the state encoding
- ❑ For example, since we have **3 outputs** (light color), encode state with **3 bits**, where each bit represents a color
- ❑ *Example states:* 001, 010, 100, 110
 - Bit₀ encodes **green** light output,
 - Bit₁ encodes **yellow** light output
 - Bit₂ encodes **red** light output
- ❑ **Minimizes** output logic
- ❑ Only works for Moore Machines (output function of state)

FSM State Encoding (III)

3. Output Encoding:

- Outputs are **directly accessible** in the state encoding

The **designer** must **carefully** choose an encoding scheme to **optimize** the design under given constraints

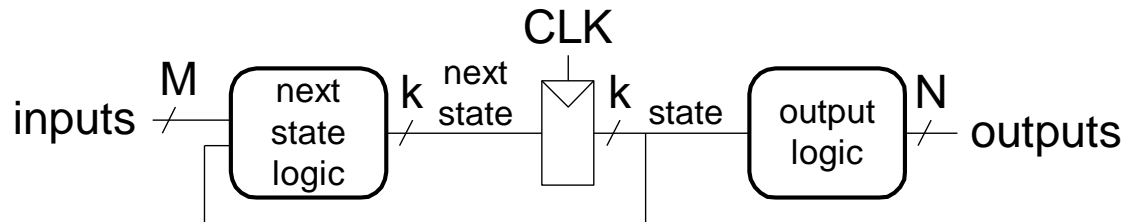
- **Minimizes** output logic
- Only works for Moore Machines (output function of state)

Moore vs. Mealy Machines

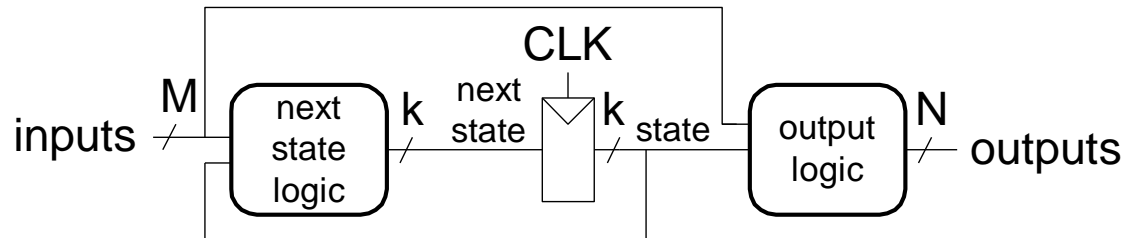
Recall: Moore vs. Mealy FSMs

- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
 - **Moore FSM**: outputs depend only on the current state
 - **Mealy FSM**: outputs depend on the current state and the inputs

Moore FSM



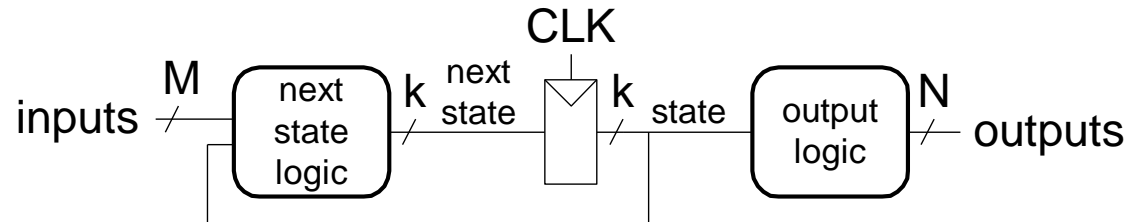
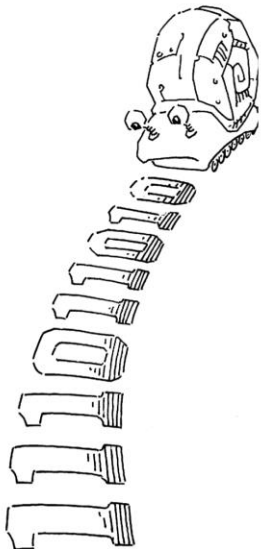
Mealy FSM



Moore vs. Mealy FSM Examples

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are **1101**.
- Design Moore and Mealy FSMs of the snail's brain.

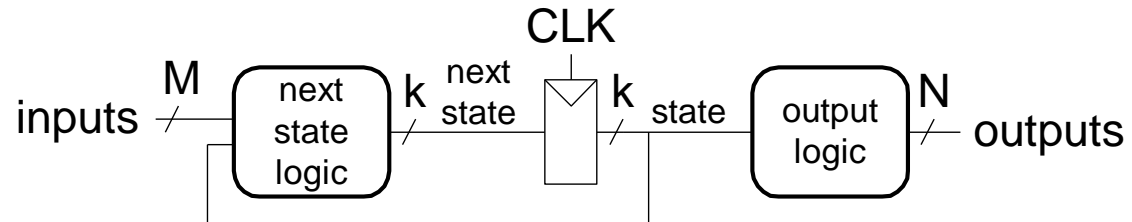
Moore FSM



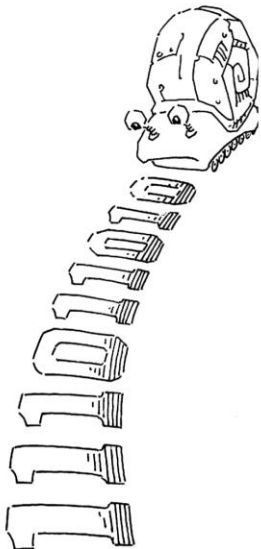
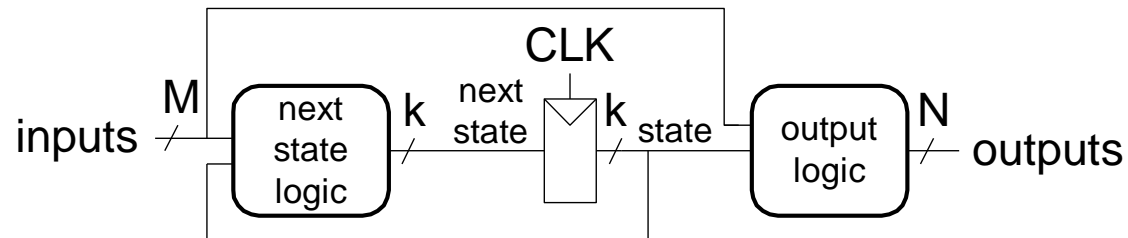
Moore vs. Mealy FSM Examples

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are **1101**.
- Design Moore and Mealy FSMs of the snail's brain.

Moore FSM

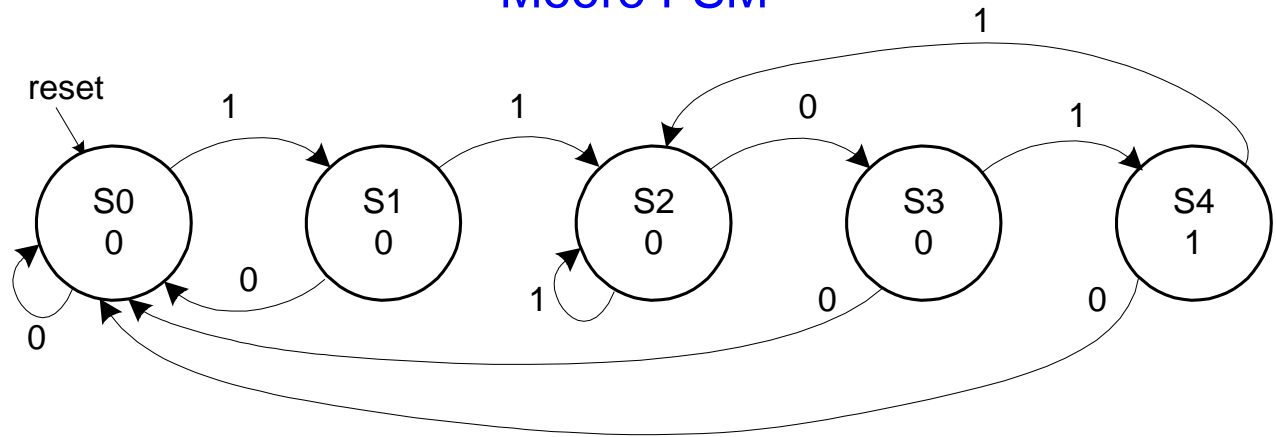


Mealy FSM



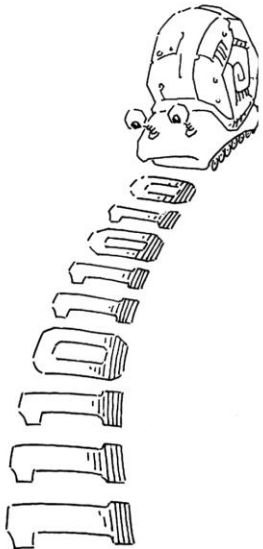
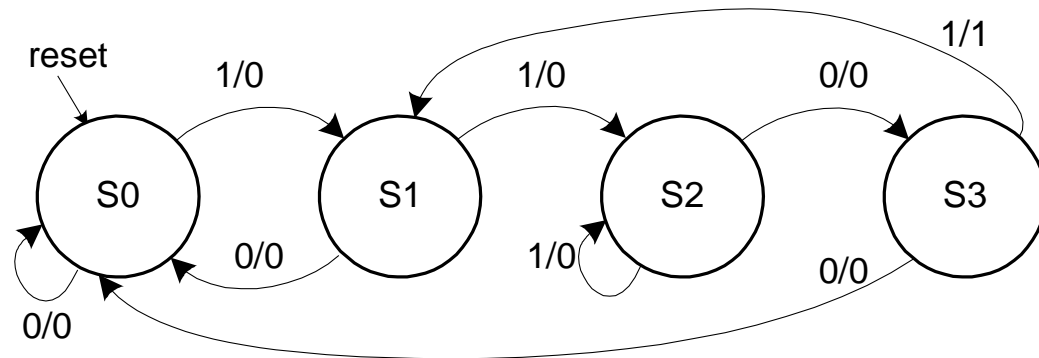
State Transition Diagrams

Moore FSM



What are the tradeoffs?

Mealy FSM



FSM Design Procedure

- **Determine** all possible states of your machine
- **Develop** a **state transition diagram**
 - Generally this is done from a textual description
 - You need to 1) determine the **inputs** and **outputs** for each **state** and 2) figure out how to get from one state to another
- **Approach**
 - Start by defining the **reset state** and what happens from it – this is typically an easy point to start from
 - Then continue to add **transitions** and **states**
 - Picking **good state names** is very important
 - Building an FSM is **like** programming (but it *is not* programming!)
 - An FSM has a sequential “control-flow” like a program with conditionals and goto’s
 - The if-then-else construct is controlled by one or more inputs
 - The outputs are controlled by the state or the inputs
 - In hardware, we typically have many concurrent FSMs

What is to Come: LC-3 Processor

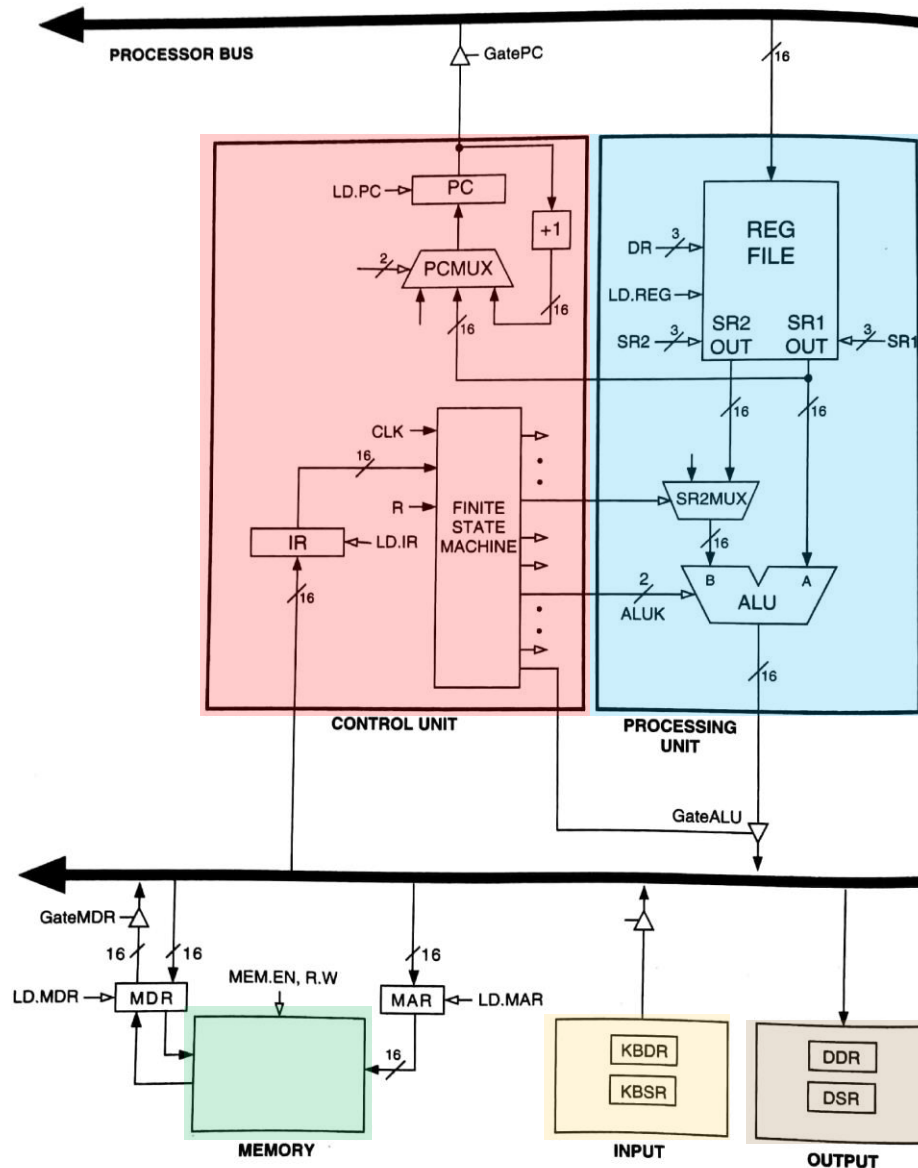
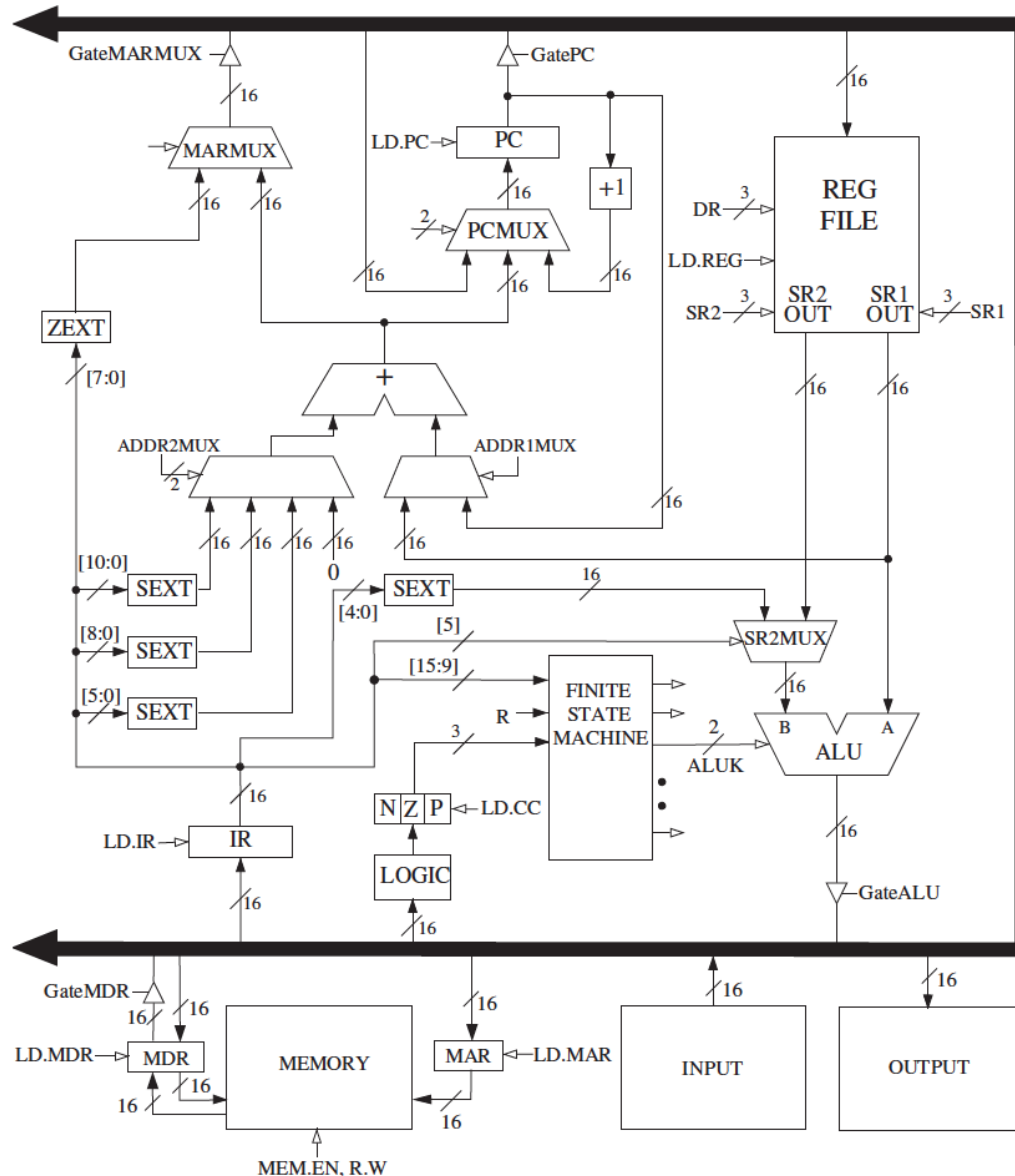


Figure 4.3 The LC-3 as an example of the von Neumann model

What is to Come: LC-3 Datapath



Design of Digital Circuits

Lecture 7.1: Sequential Logic Design II

Prof. Onur Mutlu

ETH Zurich

Spring 2019

14 March 2019

See Backup Slides:
Different Types of Flip Flops

The D Flip-Flop

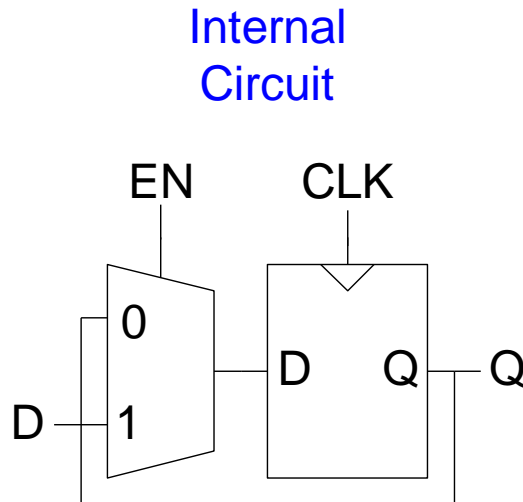
Enabled Flip-Flops

- **Inputs:** CLK, D, EN

- The enable input (EN) controls when new data (D) is stored

- **Function:**

- **EN = 1:** D passes through to Q on the clock edge
 - **EN = 0:** the flip-flop retains its previous state



Resettable Flip-Flop

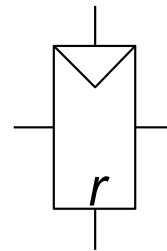
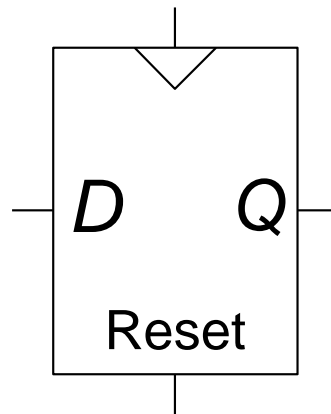
- **Inputs:** CLK, D, Reset

- The Reset is used to set the output to 0.

- **Function:**

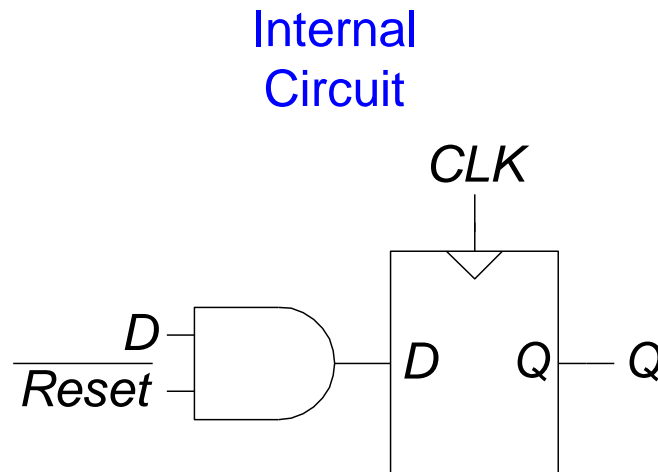
- **Reset = 1:** Q is forced to 0
 - **Reset = 0:** the flip-flop behaves like an ordinary D flip-flop

Symbols



Resettable Flip-Flops

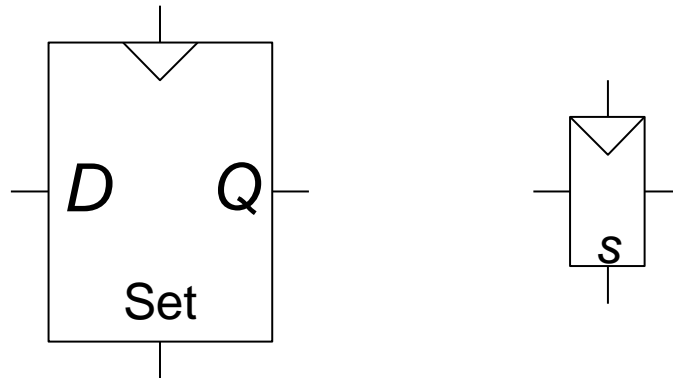
- Two types:
 - **Synchronous:** resets at the clock edge only
 - **Asynchronous:** resets immediately when Reset = 1
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop (see Exercise 3.10)
- Synchronously resettable flip-flop?



Settable Flip-Flop

- **Inputs:** CLK, D, Set
- **Function:**
 - **Set = 1:** Q is set to 1
 - **Set = 0:** the flip-flop behaves like an ordinary D flip-flop

Symbols



Recall:

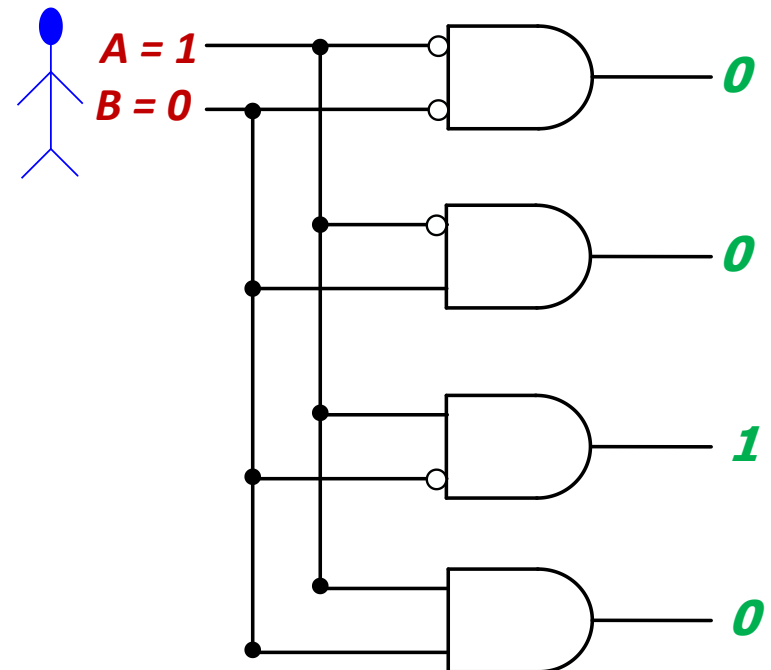
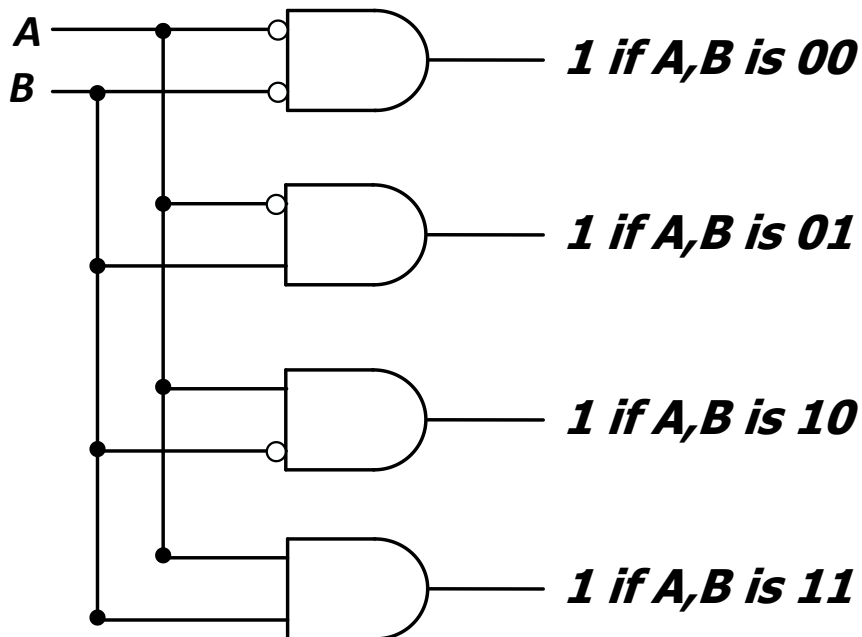
Combinational Logic Blocks

Recall: Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more **complex systems**
 - Hides the **unnecessary gate-level details** to emphasize the function of the building block
 - We now look at:
 - Decoders
 - Multiplexers
 - Full adder
 - PLA (Programmable Logic Array)
-

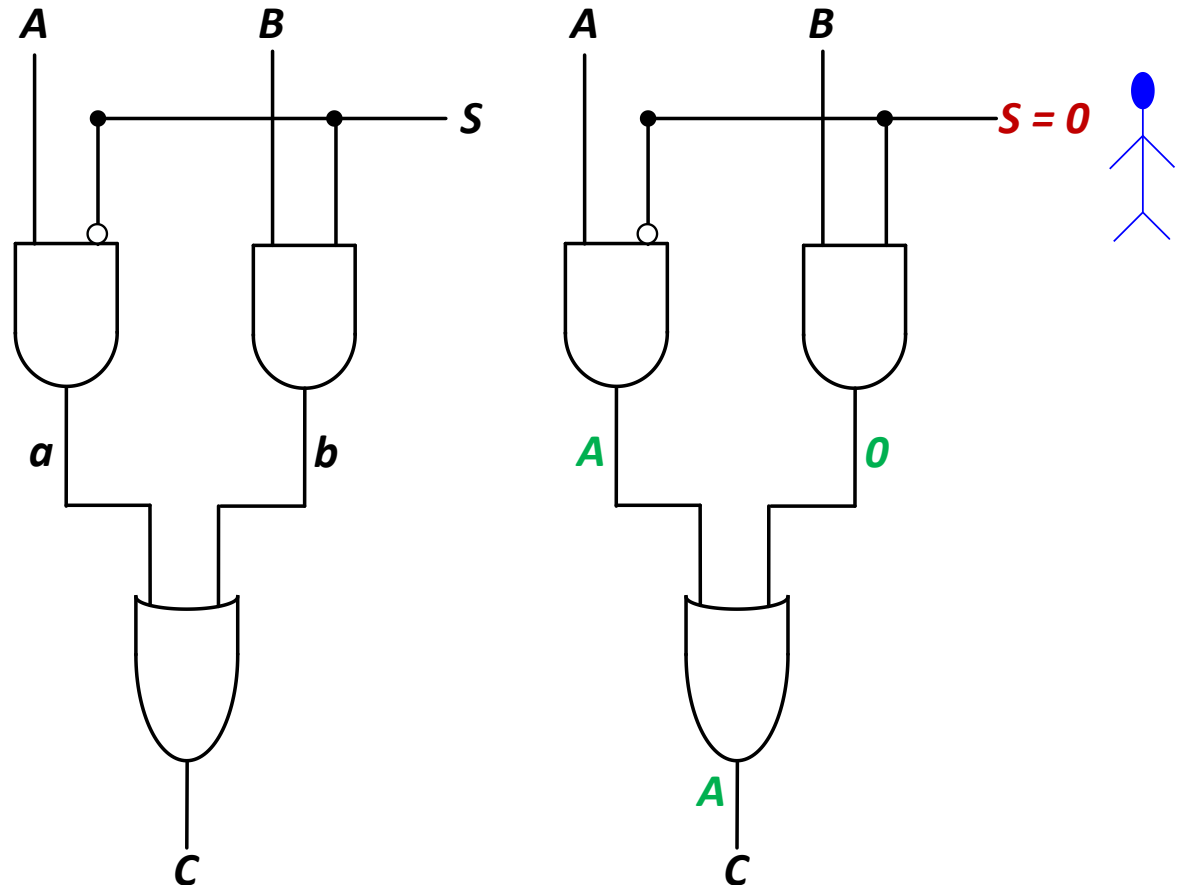
Recall: Decoder

- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The **one output** that is logically 1 is the output corresponding to the input **pattern** that the logic circuit is expected to detect



Recall: Multiplexer (MUX), or Selector

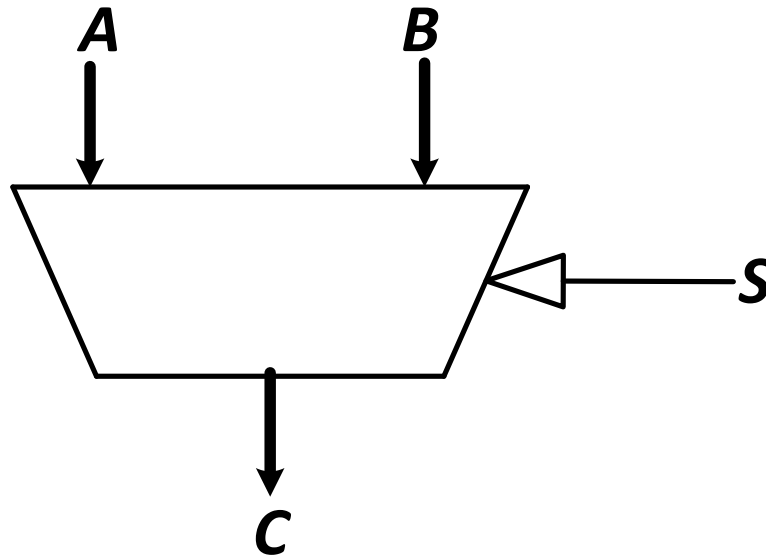
- **Selects** one of the N inputs to connect it to the output
- Needs $\log_2 N$ -bit control input
- 2:1 MUX



Recall: Multiplexer (MUX)

- The output C is always connected to either the input A or the input B
 - Output value depends on the value of the **select line S**

S	C
0	A
1	B



- **Your task:** Draw the schematic for an 8-input (8:1) MUX
 - Gate level: as a combination of basic AND, OR, NOT gates
 - Module level: As a combination of 2-input (2:1) MUXes

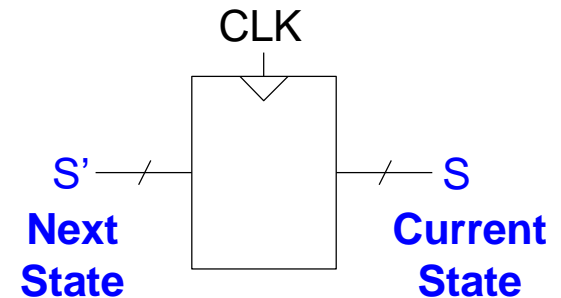
Recall:

Sequential Logic Blocks

Recall: An FSM Consists of:

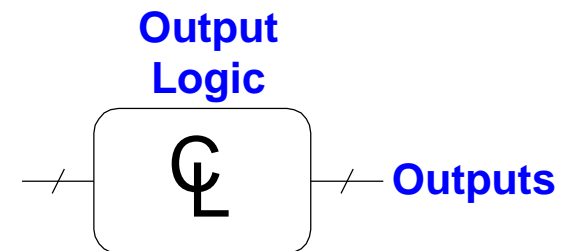
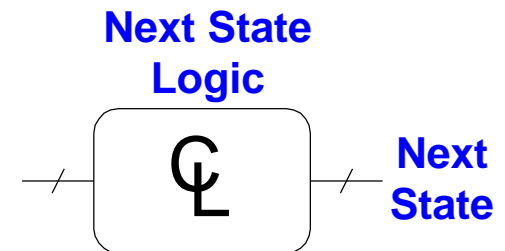
■ Sequential circuits

- State register(s)
 - Store the current state and
 - Load the next state at the clock edge



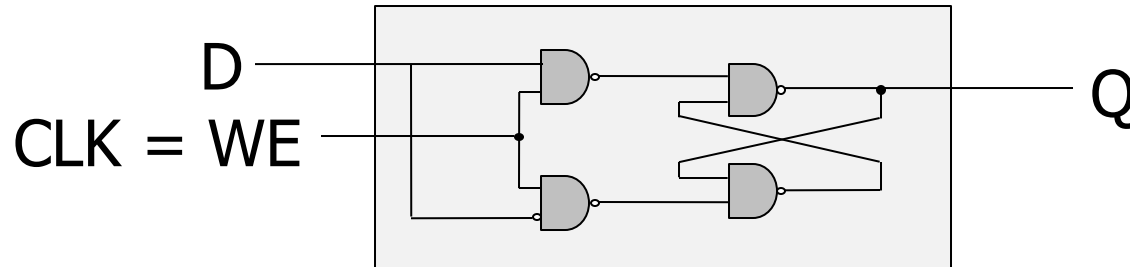
■ Combinational Circuits

- Next state logic
 - Determines what the next state will be
- Output logic
 - Generates the outputs



Recall: The Problem with Latches

Recall the
Gated D Latch



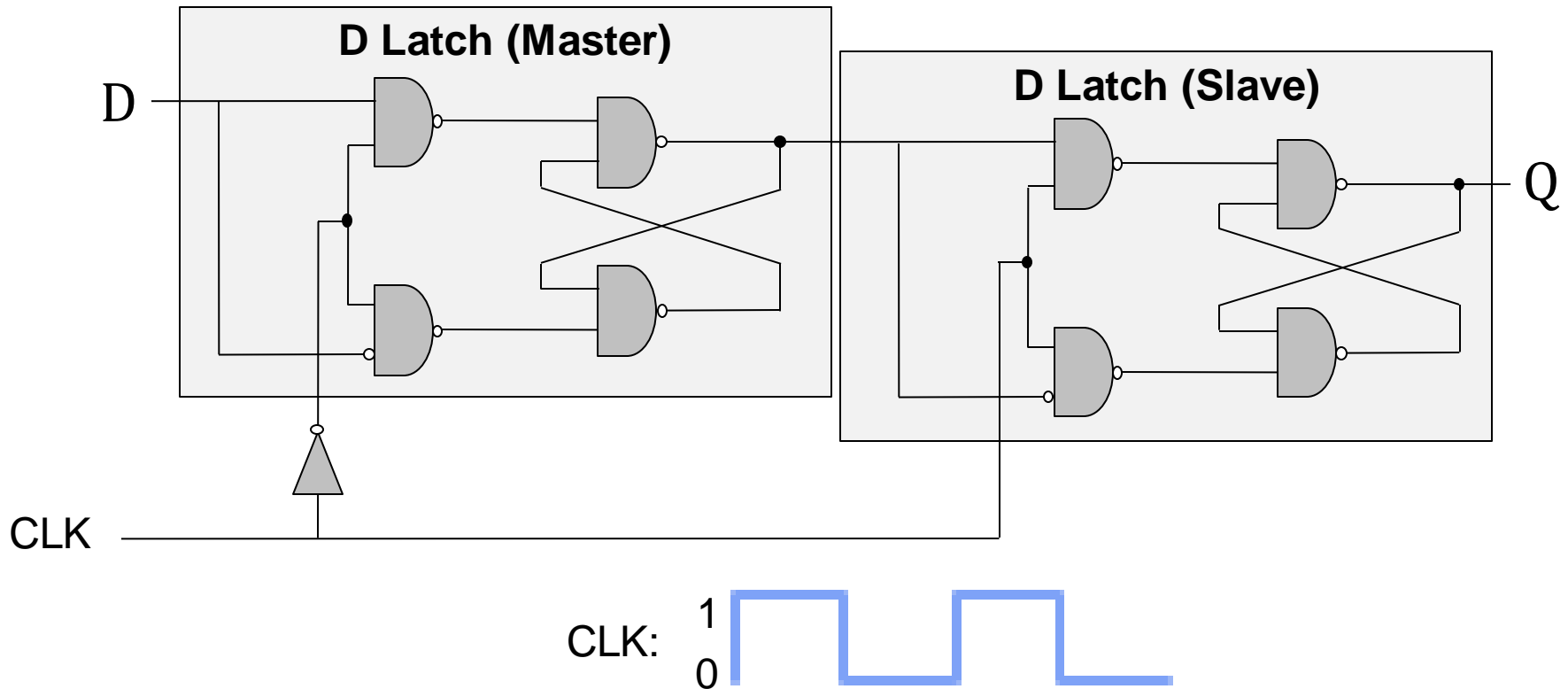
How can we change the latch, so that

1) D (input) is **observable** at **Q** (output) **only** at the **beginning of next** clock cycle?

2) Q is **available for the full clock cycle**

Recall: The D Flip-Flop

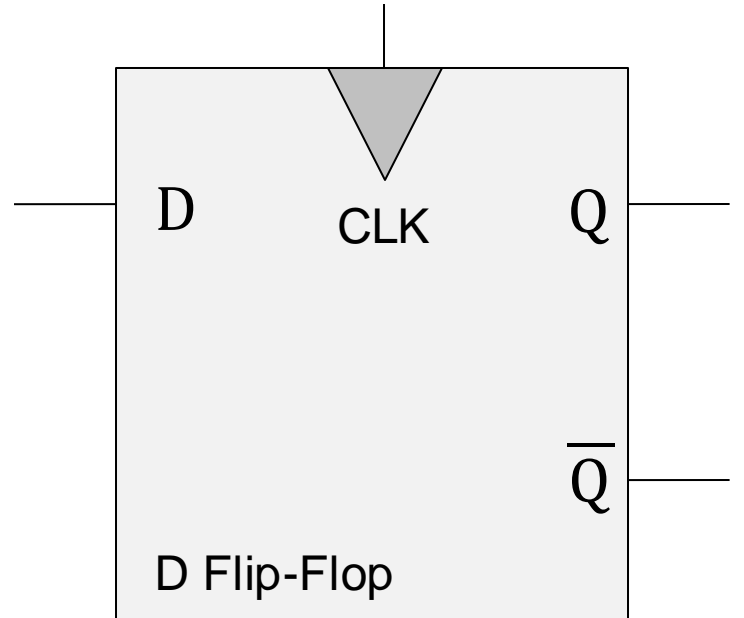
- 1) state change on clock edge, 2) data available for full cycle



- When the clock is low, master propagates **D** to the input of slave (**Q** unchanged)
- Only when the clock is high, slave latches **D** (**Q stores D**)
 - At the rising edge of clock (clock going from 0->1), **Q** gets assigned **D**

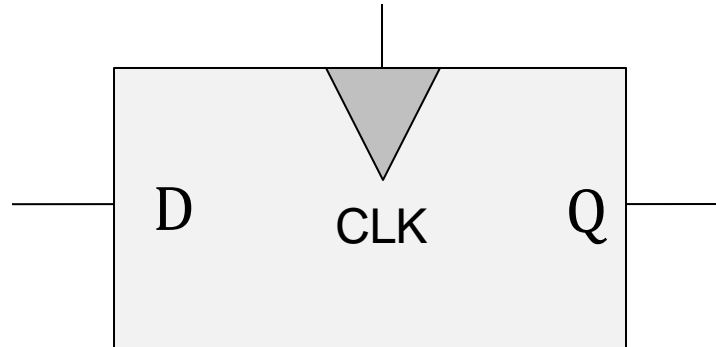
Recall: The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle



- At the rising edge of clock (clock going from 0- \rightarrow 1), **Q** gets assigned **D**
- At all other times, Q is unchanged

Recall: The D Flip-Flop



We can use these **Flip-Flops** to implement the state register!

- At the rising edge of clock (clock going from 0->1), **Q** gets assigned **D**
- At all other times, Q is unchanged